VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Vývoj subsystémů herního enginu pro FPS

Development of Game Engine Subsystems for FPS

2012 Martin Kohoutek

Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Martin Kohoutek**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Vývoj subsystémů herního enginu pro FPS
Development of Game Engine Subsystems for FPS

Zásady pro vypracování:

Cílem práce je vytvoření několika subsystémů herního enginu pro hry typu FPS. Při realizaci zadání se snažte volit co nejefektivnější postupy a algoritmy. Implementaci proveďte v jazyce C++ a dodržujte Google C++ Style Guide. Grafický subsystém bude založen na OpenGL. Zvažte také možnost využití aplikačních akcelerátorů typu NVIDIA PhysX pro vytvoření fyzikálně korektního prostředí. V textu práce popište zejména algoritmy a metody, které jste použil při řešení zadaných oblastí.

1. Seznamte se s typickými architekturami herních enginů.
2. Navrhněte následující subsystémy: generování terénu, pohyb ve vytvořeném prostředí (kolize a dynamika), vizualizace, správa zdrojů.
4. Výsledky průběžně kombinujte s paralelně vyvíjenými moduly.
5. Funkčnost celého systému ověřte na jednoduchém demu, pro které vytvořte vhodné modely.

Seznam doporučené odborné literatury:

[1] Gregory, J. Game Engine Architecture. 2009. ISBN 9781568814131.
[2] McShaffry, M. Game Coding Complete. Third edition. 2009. ISBN 1584506806.
[3] Ericson, C. Real-Time Collision Detection. 2005. ISBN 9781558607323.
[4] Millington, I., Funge, J. Artificial Intelligence for Games. Second edition. 2009. ISBN 9780123747310.
[5] Wright, R. S., Lipchak, B., Haemel, N. S. OpenGL(R) SuperBible: Comprehensive Tutorial and Reference. Fifth edition. 2010. ISBN 0321712617.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Tomáš Fabián**

Datum zadání: 18.11.2011

Datum odevzdání: 04.05.2012

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 4. května 2012 .....

Chtěl bych poděkovat Ing. Tomáši Fabiánovi za věcné připomínky, návrhy, udání směru této práce a podněty k zamyšlení během jejího vypracovávání.

Abstrakt

Tématem této bakalářské práce je vývoj herního subsystému. To je rozděleno do více částí, počínaje generováním samotného prostředí, kde se veškeré akce budou odehrávat. V tomto případě se jedná o exteriérové prostředí, konkrétně část terénu. Další, nejdůležitější částí je problematika efektivního vykreslování. Vykreslit jen to co má být vykresleno a to co nejrychleji. Poslední částí práce je vzhled a tudíž celkový dojem virtuálního světa. Textury, detaily, osvětlení a stíny jsou to co upoutá jako první a také jedna z hlavních věcí, co herní systém prodává. Problém optimalizace, vzhledu aplikace a fyziky modelů jsou aspekty, které, když jsou dobře odvedené, dělají kvalitní herní systém.

Klíčová slova: FPS subsystém, efektivní vykreslování, úroveň detailů, generování terénu

Abstract

The main idea if this bachelor's thesis is the development of game sub-engine. It is divided into several parts, starting with generation of environment where every action will take place. In this case it is exterior environment, specifically part of terrain. Another, the most important part is efficient render issue. Draw only what has to be rendered and as fast as possible. The last part of this thesis is appearance and whole impression of virtual world. Textures, details , lighting and shadows are the most attracting and the most important parts that solds gaming system. The problem of optimalization, appearance of application and models physics are aspects, if done well, that do perfect gaming engine.

Keywords: FPS sub-engine, effective render, level of detail, terrain generation

Seznam použitých zkratek a symbolů

GLSL	–	OpenGL Shading Language
LOD	–	Level of detail
OQ	–	Occlusion query
HDR	–	High dynamic range
FPS	–	First person shooter
BSP	–	Binary space partitioning
AABB	–	Axis-aligned bounding box
HOM	–	Hierarchical occlusion map
VAO	–	Vertex Array Object
VBO	–	Vertex Buffer Object
IBO	–	Index Buffer Object
TBO	–	Texture Buffer Object
UBO	–	Uniform Buffer Object
DLOD	–	Discrete level of detail
CLOD	–	Continuous level of detail

Obsah

Seznam obrázků

1 Úvod

Počítačová grafika je nedílnou součástí informačních technologií, ale i dnešního světa vůbec. Její využití je od „pouhého“ zobrazení aplikace uživateli, až po vytváření dokonalých, téměř realistických, snímků nebo videí. Bez tohoto odvětví bychom si asi těžko představili většinu dnešních filmů nebo i reklam. Její největší využití avšak sledujeme v herním průmyslu.

Na rozdíl od filmů, kde je děj daný a nezáleží na optimalizacích, musí hry dynamicky řešit akce uživatele a poskytovat mu odezvu. Také záleží na rychlosti výpočtu každého snímku. Od videí, kde se jeden snímek může počítat i desítky minut, se hry liší tím, že zde by to mělo být maximálně 1/30 vteřiny, aby byla aplikace plynulá a nedocházelo k tzv. „zamrzání“.

Přestože výkon dnešních počítačů se může zdát dostatečný, je třeba brát v úvahu postupy, řešící optimalizaci vykreslování každého snímku. Rychlost vykreslování, dána počtem vykreslených snímků za vteřinu (ang. framerate), je důležitým parametrem herního systému. Dalším aspektem, který už může uživatel zpozorovat, je výsledný vzhled aplikace (kapitola 4.3.2).

Optimalizace vykreslování byla hlavní částí této práce. Je nutné nepočítat s objekty, které leží mimo pohled kamery, nepočítat s objekty zakrytými jinými objekty a vzdálené objekty zobrazovat v menší kvalitě než ty nejbližší. O těchto optimalizacích dále v kapitole 5.Efektivní vykreslování a vzhled tvoří kvalitní herní systém (ang. game engine). V kapitole 2 si představíme nejznámější herní systémy. Podíváme se jak na starší, tak i na nové, nejžádanější systémy.

Výběrem tohoto tématu bych si chtěl prohloubit stávající znalosti v počítačové grafice, neboť je to pro mě zajímavé téma. Očekávám, že lépe porozumím problematice vykreslování a její optimalizaci. Také si chci ověřit znalosti z předmětů týkajících se počítačové grafiky na rozsáhlejším příkladu. Věřím, že tato práce změní můj pohled na herní odvětví a budu vědět, co všechno se skrývá za vyřešením problému, který běžný uživatel ani nevnímá.

2 Herní systémy

Herní systém je hlavní jádro hry, které zajišťuje její funkcionalitu. To obsahuje části pro vykreslování, herní fyziku, hierarchii scény, animace, zvuky a další. Přední herní systémy poskytují rozhraní pro vytváření her, čímž se vývoj her urychluje.

Termín herní systém vznikl v polovině 90. let ve spojení s 3D hrami typu FPS (z ang. first person shooter). V této době vznikly legendární hry `Doom` a `Quake`. Tyto hry byly vytvářeny jiným postupem než ostatní hry. Část vývojářů poskytlo jádro hry, které se staralo o samotnou funkčnost, a další část navrhla vlastní vzhled, postavy, zbraně a úrovně, tvořící herní obsah. Oddělením pravidel a obsahu od jádra (engine) aplikace umožnilo zaměření různých týmů na konkrétní problémy.

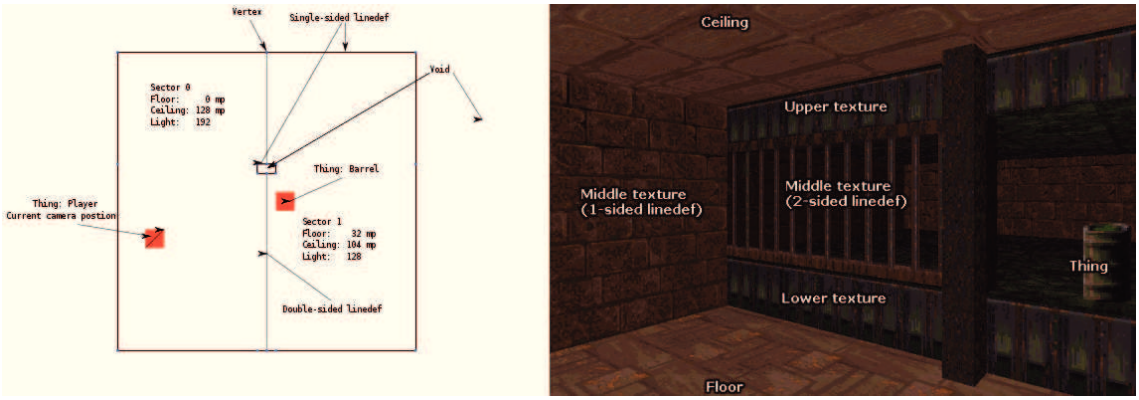
Později vytvořené hry jako `Quake 3 Arena` nebo `Unreal` stály na stejném konceptu oddělení engine od herního obsahu. Engine lze tak použít pro více aplikací bez nutnosti programování celé funkčnosti znovu. Tím se usnadňuje a hlavně urychluje vývoj celé hry. Navíc licencování takové technologie se prokázalo jako užitečné. Ceny licencí nejlepších herních systémů sahají do milionů dolarů za licenci. Počet držitelů této licence může být až několik desítek společností, jak je tomu i u `Unreal engine`.

2.1 Přehled herních systémů

Nejznámější herní systémy, které udaly nový trend vývoje her.

2.1.1 Doom engine (1993)

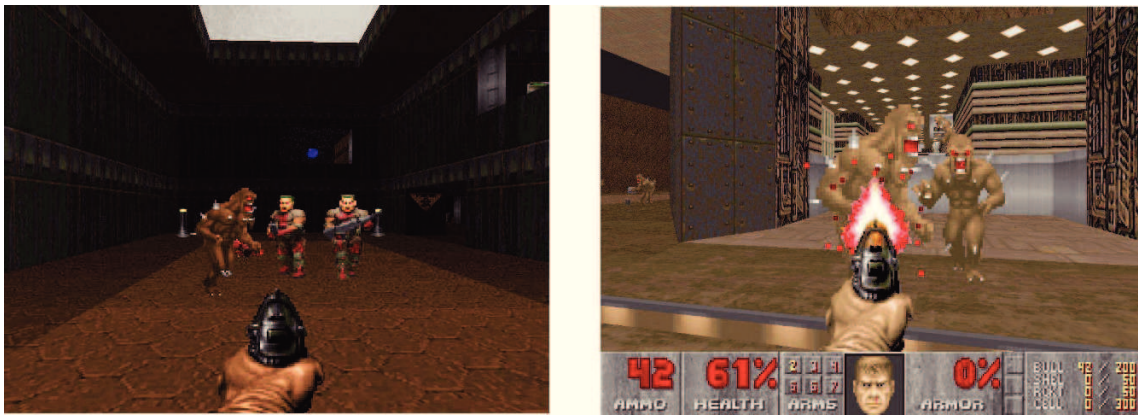
Herní engine společnosti `id Software` zvaný `id Tech1`. Všechny úrovně jsou z vrchního pohledu pouze dvourozměrné. Nevýhodou bylo, že nemohly být dvě patra nad sebou, ale výhodou bylo jednoduché zobrazení mapy. Mapa byla rozdělena do místností zvaných sektory. Prostory byly ohraničeny liniemi, které měly vždy buď jednu nebo dvě strany. Jedna strana představovala zeď ohraničující mapu, dvě znamenaly přepážku mezi sektory.



Obrázek 1: Zobrazení mapy a její 3D reprezentace z pohledu hráče (převzato z [13])

Hierarchie scény byla vytvořena pomocí BSP (angl. binary space partitioning) stromu. Je to proces dělení prostoru pomocí roviny. Prostor je rozdělen na dvě části a každé je přiřazena skupina objektů, které v ní leží. Vzniklé části jsou rekurzivně děleny nově zvolenou rovinou na 2 části. Tímto algoritmem se rozdělila scéna před začátkem úrovně. Tímto ale vznikl problém. Dveře a výtahy se pohybovaly pouze nahoru a dolů, nikoliv do stran.

Hry využívající doom engine: `Doom (1993), Doom II (1994), Hexen (1995)`



Obrázek 2: Ukázky z her Doom a Doom 2

2.1.2 Quake engine (1996)

Herní engine společnosti `id Software` zvaný také `id Tech 1`, později vylepšen na `id Tech 2 (Quake II engine)`. Používal BSP strom pro dělení scény. Pro pohybující objekty využíval Gouraudovo stínování, pro statické objekty „světelnou mapu“ ovlivňující jas výsledné plochy. Quake byla první pravá 3D hra používající speciální mapový systém pro předvýpočet a předkreslení 3D mapy. Po vypočtení mapy se provedl výpočet a připravení osvětlovacích map pro ulehčení práce procesoru při hraní.

Další optimalizací bylo odstranění částí prostoru, které nebyly momentálně vidět a zabránění jejich výpočtu. Na tento problém byla využita technika Z-bufferu. Provedl se výběr nejbližšího polygonu a zahodily se polygony ukryté za tímto. Pokud hráč neviděl do nějaké místnosti, znamenalo to, že engine nemá počítat s objekty v této části. Proto se ve hře objevovaly pravoúhlé tunely vedoucí z jedné velké místnosti do druhé. Quake byla jedna z prvních her využívajících hardwarovou 3D akceleraci. Podporoval hru pro více hráčů po LAN nebo v internetu.

Hry využívající quake engine: `Quake (1996), Hexen II (1997), Half-Life (1998)`

Obrázek 3: Ukázka hry Quake

2.1.3 Unreal engine (1998-)

Konkurencí `id Softwaru` byla společnost `Epic Games`. Vyvinuli engine primárně navržený pro FPS hry s názvem `Unreal engine`. Velká část programu může být napsána v `UnrealScript` skriptovacím jazyce bez nutnosti hlubšího zkoumání vnitřního fungování engine. Engine je napsán v `C++, C#, UnrealScript, HLSL, GLSL, CUDA` a je multiplatformní (OpenGL a DirectX).

Unreal engine 1 První generace `Unreal engine` vytvořena v roce 1998. Bylo třeba udělat pár ústupků (jednodušší detekce kolizí) pro plynulý chod systému na počítačích té doby. Také síťový protokol pro hru více hráčů nebyl zpočátku tak dokonalý jako u konkurenčního `id Tech2` systému. Popularita tohoto herního systému se skrývala v jednoduchém vývoji dalších rozšíření. Tento engine byl první, který implementoval pravou klient-server architekturu.

Unreal engine 2 V roce 2002 došlo ke kompletnímu přepsání jádra systému a postupů vykreslování. Přidán nový editor úrovní a fyzikální nástroje, využité při animaci padajících postav (ang. ragdoll physics) nebo simulaci vozidel. Pro editor map byl přidán editor částic a podpora pro 64-bitové systémy u hry `Unreal Tournament 2004`.

Unreal engine 3 Systém navržený pro široké použití od osobních počítačů, přes konzole jako PlayStation 3, Xbox 360, Wii, až po Android a OpenGL operační systémy jako iOS nebo MacOS X. Využíval pokročilé techniky vykreslování jako větší dynamický rozsah expozice (HDR), výpočet osvětlení pro každý pixel obrazu (ang. per-pixel lighting) a dynamické stíny. Získal podporu od mnoha velkých společností jako jsou `2K Games, Electronic Arts, Konami, Midway Games, Sega, Sony, THQ, Ubisoft`.

Unreal engine 4 Vyvíjen od roku 2003. Představen by měl být tento rok (tj. 2012).

Hry založené na Unreal engine: `Unreal Tournament, BioShock1/2, DeusEx, Batman:Arkham Asylum, Gears of War1-3, Mass Effect1-3`

Obrázek 4: Ukázka hry Unreal Tournament (1999)

Obrázek 5: Ukázka hry Batman: Arkham Asylum (2009)

2.1.4 CryEngine

Začínal jako technologické demo od společnosti Crytek pro nVidii. Ta v systému viděla budoucnost a nakonec došlo k vytvoření hry na tomto systému. Tak v roce 2004 vznikl CryEngine 1 a byla vytvořena hra s názvem `Far Cry` s ohromující grafickou stránkou a téměř neomezenou volností pohybu. Využíval HDR expozici a jednalo se o první systém s per-pixel stínováním na trhu.

V roce 2007 poznal svět hru `Crysis`, která byla poháněna dalším stupněm - CryEngine 2. Osvětlení v reálném čase, dynamické měkké stíny, vzdálená mlha, pokročilé shadery, kvalitní 3D oceán a prosvítání paprsků skrz vegetaci jsou technologie, na které se vývojáři zaměřili při vytváření tohoto systému.

Poslední verzí je CryEngine 3 (2009) představovaný jako nejrychlejší přední engine na světě se specifickými vlastnostmi pro PC, Xbox360 a PlayStation 3. Systém založený na realisticky vypadajících postavách a jejich dokonalých animacích, které celkem inteligentně reagují na okolní prostředí pomocí úhlu pohledu a sluchu postavy. Na systému založená hra `Crysis 2` je poslední hrou od firmy Crytek. Další hrou, založenou na tomto engine bude hra `Crysis 3`, která vyjde v roce 2013. Crytek na svých stránkách zdarma nabízí CryEngine pro nekomerční vytváření her.

Obrázek 6: Ukázka hry Crysis 2 (2011)

3 Generování terénu

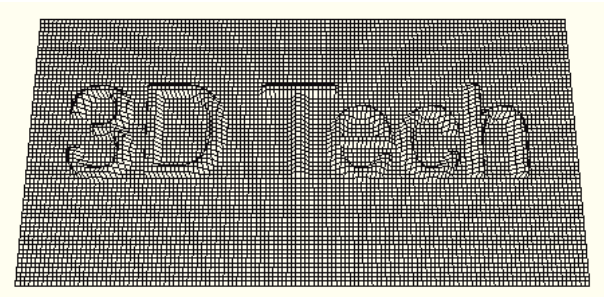
Generování terénu je rozsáhlé téma, které nemá přesně dané řešení. Dva základní přístupy vytvoření terénu jsou:

- **Výškové mapy**
- **Umělé vytváření terénu**

3.1 Výškové mapy

Výškové mapy jsou první věc, kterou bereme v úvahu při vytváření terénu, protože jsou jednoduše aplikovatelné. Jsou reprezentovány nejčastěji černobílým obrázkem. Každý pixel obrázku má hodnotu od 0 (černá) do 255 (bílá). Tuto reprezentaci si převedeme do pole s hodnotami každého pixelu. V aplikaci nám poté stačí na reprezentaci terénu vygenerovat 2D síť se souřadnicemi X a Z. Hodnoty sítě procházíme a každé přiřadíme výšku (souřadnici Y) z pole výškové mapy. Tím vznikne terén odpovídající obrazu výškové mapy.

Obrázek 7: Příklad výškové mapy (převzato z [15])

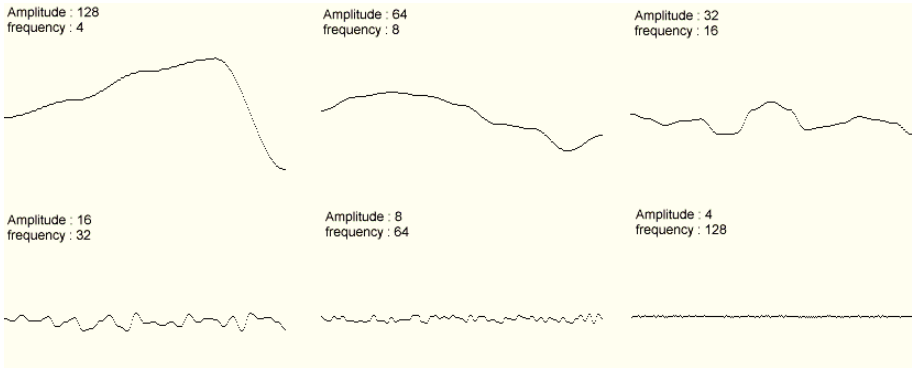


Obrázek 8: Vytvořený profil z výškové mapy (převzato z [15])

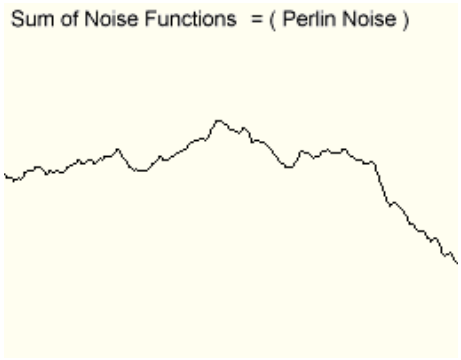
3.2 Umělé vytváření terénu

3.2.1 Vygenerování základního šumu

Přesvědčivěji vypadající terén se vytváří náhodným generováním pole hodnot a jeho dalším zpracováním. Tímto generováním vzniká tzv. šum, který by se při uložení do obrázku podobal výškové mapě. Nejčastěji se používá `Perlin noise`. Je to příklad šumu, který se skládá z více šumů složených do jednoho výsledného.



Obrázek 9: Vygenerované šumy pro vytvoření výsledného šumu (převzato z [14])

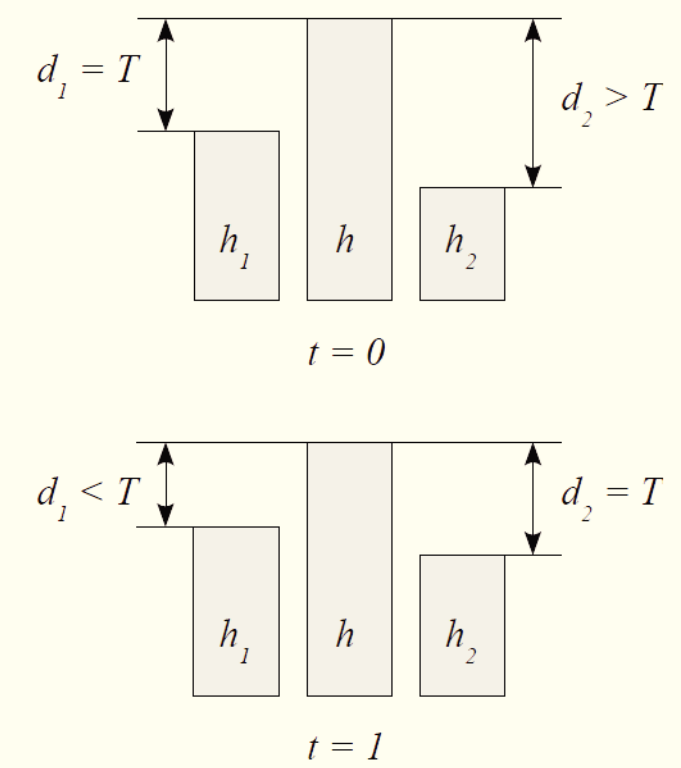


Obrázek 10: Výsledný `perlin noise` (převzato z [14])

Šumy jsou vygenerovány s různými amplitudami a frekvencemi. Každý další šum má poloviční amplitudu a dvojnásobnou frekvenci, Každý šum je zván oktáva. Pro hladký terén musí být výsledný šum složen z oktáv, kde jsou při nízkých frekvencích vysoké amplitudy a u vysokých frekvencí malé amplitudy. U zvrásněného terénu je to naopak. Je tedy jasné, že frekvence oktávy, která má největší amplitudu ovlivňuje výsledný terén nejvíce. Amplituda při každé oktávě se nazývá perzistence, termín podle Mandelbrota. Další možností generování terénu jsou Voronoi diagramy nebo technika `midpoint displacement`.

3.2.2 Dotváření realistického terénu

Vygenerovaný základní šum by mohl být postačující pro reprezentaci terénu. V praxi se však využívá ještě erozních algoritmů a vyhlazení terénu pro co nejrealističtější výsledek. Erozní algoritmy mají představovat klasickou erozi jako v reálném světě.



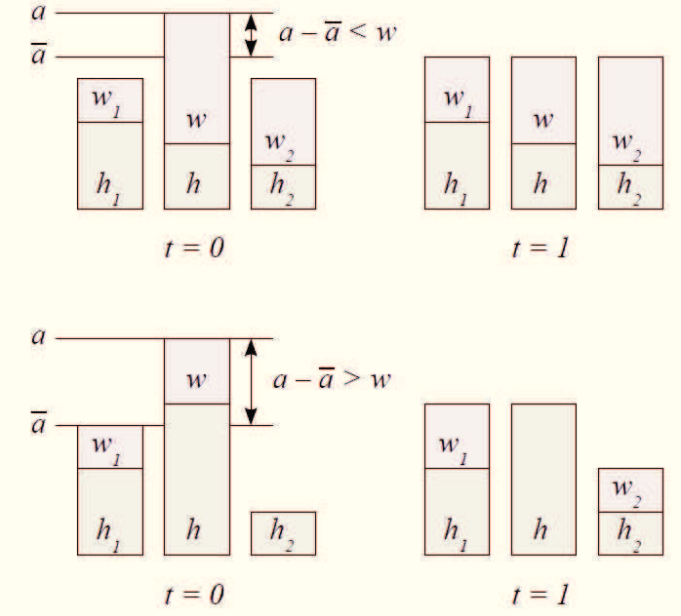
Obrázek 11: d2 je větší, než referenční hodnota, proto je materiál přesunut z h na h2 (převzato z [5])

Termální eroze spočívá v procházení terénu a zjišťování rozdílu výšek mezi aktuálně procházeným bodem a jeho čtyřmi sousedními body terénu. Když je rozdíl výšek větší než předem daná maximální hodnota, je z vyššího bodu odebrána část výšky, která se přidá menšímu bodu, aby rozdíl výšek byl menší než referenční hodnota.

Hydraulická eroze přidává ke každému bodu terénu část vody. Udržují se informace o mapě s vodní hladinou a o mapě se sedimenty.

1. Každému bodu se přidá konstantní hodnota vody
2. Část výšky terénu proporcionálně k množství vody se převede na sediment
3. Voda a sediment každé výšky se přesouvá mezi sousedními body jako u termální eroze

4. Část vody zmizí a část sedimentu, který překročil maximální hodnotu, kterou může voda nést se přičte zpět k dané buňce



Obrázek 12: Dvě možnosti hydraulické eroze. V prvním případě je přesunuta voda na vyrovnání hladiny, ve druhém je přesunuta veškerá voda, aniž by bylo dosaženo referenční výšky (převzato z [5])

Pro generování terénu jsem zvolil `perlin noise` jako základ generování. Poté jsem použil Termální erozi pro nejvyšší části mapy. Celou mapu jsem podle výšky vyhladil. Pro vyhlazení jsem použil tuto funkci:

```
for( int x = 1; x < 1024; x++ ) {
    for( int z = 1; z < 1024; z++ ) {
        heightmap[x][z] = (1.0/9.0)*heightmap[x−1][z−1] + (1.0/9.0)*heightmap[x][z−1] + (1.0/9.0)*
            heightmap[x+1][z−1] +
                                (1.0/9.0)*heightmap[x−1][z] + (1.0/9.0)*heightmap[x][z] +
                                    (1.0/9.0)*heightmap[x+1][z] +
                                (1.0/9.0)*heightmap[x−1][z+1] + (1.0/9.0)*heightmap[x][z+1] +
                                    (1.0/9.0)*heightmap[x+1][z+1];
    }
}
```

Výpis 1: Funkce vyhlazení terénu

4 Rozhraní pro vykreslování

Protože je třeba aplikaci určitým způsobem vykreslit a tak ji zobrazit uživateli, je třeba zvolit rozhraní pro vykreslování. Zvolil jsem si OpenGL (Open Graphics Library) knihovnu před konkurenční DirectX. Tato knihovna byla vytvořena firmou `Silicon Graphics` v roce 1992. Je multiplatformní a začátky s ní jsou jednodušší a návodů na internetu je také o něco více. Její velká popularita je částečně zásluhou kvalitní dokumentace.

4.1 Využití nejnovějších technologií

Poslední verzí OpenGL je verze 4.2 (8. srpna 2011). Avšak již při vypuštění OpenGL verze 3.0 do světa (2008) nastal zlom v celkovém fungování programu. Přesto se neustále setkáváme se starou verzí 2.x. Tutoriálů a programů napsaných v této verzi grafické knihovny koluje po internetu spousta. Vzhledem k použití v budoucnosti je to ovšem spíše krok vzad. OpenGL se posledními verzemi implementací dost přiblížilo DirectX knihovně od Microsoftu, která je hojně využívaná v herním průmyslu.

OpenGL verze 3.x a výše se zcela od základu fungování změnila. Funkcionálně se OpenGL dostalo na úroveň DirectX a případný přechod z jedné knihovny na druhou je nyní jednodušší. Každá z knihoven postrádá pár funkcí z konkurenční, ale z celkového pohledu jsou obě knihovny v dnešní době na přibližně stejné úrovni. OpenGL je navíc multiplatformní.

OpenGL 3.x již nemá danou funkční pipeline. Neexistují zde bloky `glBegin`, `glEnd` a věci s tím spojené jako `glVertex` atd. K práci s vertexy je tedy nutné použít VBO pro uchovávání informací o vertexech (body tvořící objekt). K vykreslování je třeba použít shadery (vice v kapitole 4.3.2). Od verze 3.3 se srovnalo číslování OpenGL a GLSL (dále v kapitole 4.3.2).

Podstatné nové vlastnosti v OpenGL 3.x:

- **Transform feedback** - možnost zapisovat výsledek transformace shaderem do vertex bufferu
- **TBO** - kontejner určený pro textury
- **UBO** - pro tzv. uniforms datové typy používané shadery
- **VAO** - kontejner pro VBO, IBO
- **Instancing** - vykreslení jednoho objektu na několik pozic s využitím stejných dat
- **Geometry shadery** - ovlivňují výslednou geometrii

OpenGL 3.0 zavedlo myšlenku o tzv. „nedoporučování“ používání staršího OpenGL kódu. To vedlo k odebrání těchto zakázaných funkcí z jádra OpenGL verze 3.1. Některé implementace si žádaly použití starších funkcí a tak se zavedlo rozdělení na dva módy:

- **Core** - pro funkce obsažené v jádru OpenGL verze

- **Compatibility** - pro zpětnou kompatibilitu se staršími funkcemi

Tyto módy jsou využívány při vytváření kontextu OpenGL, což je, jednoduše řečeno, okno k vykreslování nutné k používání OpenGL. Ve verzi 3.0 šlo vybrat mód kontextu, ale dostupný byl jen jeden `Core`. Režim `Compatibility` byl přidán až ve verzi 3.2.

```
// glut inicializace – OpenGL API
glutInit ( &argc, argv ) ;
// nastavení zobrazovacího módu
glutInitDisplayMode( GLUT_DEPTH | GLUT_DOUBLE | GLUT_RGBA );

// vytvoření OpenGL kontextu verze 4.2
glutInitContextVersion ( 4, 2 ) ;

// nastavení CORE módu
glutInitContextFlags ( GLUT_CORE_PROFILE | GLUT_DEBUG );
   // GLUT_COMPATIBILITY_PROFILE – pro kompatibilni mód

// nastavení velikosti okna
glutInitWindowSize( screen_width, screen_height ) ;
// nastavení titulku okna
glutCreateWindow( "OpenGL_4.2" );

// glew inicializace – pro podporu rozšíření (VBO,VAO, ...)
glewExperimental = GL_TRUE;
glewInit () ;
```

Výpis 2: Ukázka vytvoření kontextu při inicializaci OpenGL okna

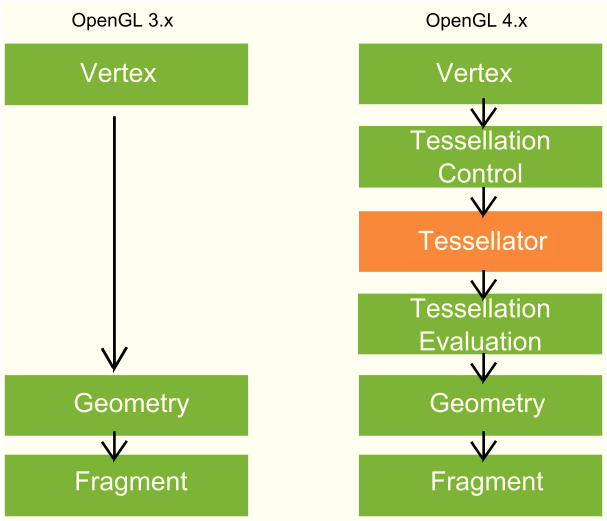
Kontext verze 3.0 a výše může být označen bitem `Forward compatibility`. Tento bit zajistí, že funkce označené jak zastaralé budou odstraněny. Bit může být použit ve spojení s `Core` i `Compatibility` kontextem. Pro jednotlivé verze to znamená:

- **3.0** - všechny zastaralé funkce nebudou použitelné
- **3.1** - zbylé funkce (zastaralé pro 3.0, ale neodstraněné v 3.1) budou odstraněny
- **3.2+ compatibility** - neodstraní nic, protože v módu kompatibility nejsou žádné funkce označené jako zastaralé
- **3.2+ core** - odstraní zastaralé funkce (široké čáry)

OpenGL verze 4.x přišla s dalšími vlastnostmi. Jako nejzajímavější se jeví HW teselace. Je to proces, kdy je vstupní polygon rozdělen na více částí a vzniká tím detailnější model. Využití najdeme při implementaci úrovně detailů prostředí (6), při zkvalitnění modelu objektu pro metodu displacement-mapping, což je technika změny povrchu a dodání detailů. Další využití je pro vektorovou grafiku nebo vykreslování vlasů a srsti.

4.2 Teselace

Teselace přidala další tři stupně do OpenGL shader pipeline (průchod vertexů shadery). Jsou to programovatelné `Control shader`, `Evaluation shader` a pevně daný generátor primitiv `Tessellator`. Přidán nový typ primitiv - `GL_PATCHES`. Můžeme definovat, kolik bodů tvoří jeden `patch` (možnost 1 až 32). Na obrázcích je rozdíl OpenGL 3.x a OpenGL 4.x pipeline.



Obrázek 13: Rozdíl v shader pipeline OpenGL 3.x a OpenGL 4.x (převzato z [7])

4.2.1 Control shader

Spuštěn jednou pro každý vertex. Počítá vnitřní a vnější LOD pro každý `patch`.

```
layout(vertices = 3) out;

uniform float tessLevelOuter;
uniform float tessLevelInner;

void main()
{
    gl_TessLevelOuter[0] = tessLevelOuter;
    gl_TessLevelOuter[1] = tessLevelOuter
    gl_TessLevelOuter[2] = tessLevelOuter;
    gl_TessLevelInner[0] = tessLevelInner;
    gl_out[gl_InvocationID].gl_Position = gl_in[gl_InvocationID].gl_Position;
}
```

Výpis 3: Ukázka control shaderu

4.2.2 Tessellator

Používá levely teselace pro vytvoření detailní sítě s novými vertexy.

4.2.3 Evaluation shader

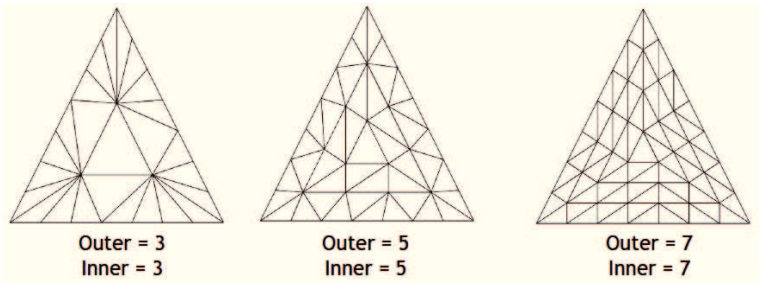
Počítá pozici každého vertexu vytvořeného teselátorem. Lze zadat směr generování bodů (cw,ccw).

```
layout(triangles, equal_spacing, ccw) in;

void main()
{
    gl_Position = vec4(gl_In[0].gl_Position.xyz* gl_TessCoord.x+
                       gl_In[1].gl_Position.xyz* gl_TessCoord.y+
                       gl_In[2].gl_Position.xyz* gl_TessCoord.z+
                       1.0);
}
```

Výpis 4: Ukázka evaluation shaderu

Příklady teselace Výsledky teselace trojúhelníku jako vstupního primitiva do teselátoru. Parametry nastavované u teselace jsou parametry `gl_TessLevelOuter` a `gl_TessLevelInner` control shaderu.



Obrázek 14: Teselace trojúhelníku pro různé úrovně teselace (převzato z [7])

4.3 Vertex buffery a shadery

Jak již bylo řečeno OpenGL verze 3.x pro uložení objektů musíme použít vertex buffery a data zpracovat pomocí shaderů.

4.3.1 Vertex buffery

Jsou to konstrukce, které nejsou uloženy v systémové paměti, ale jsou v paměti GPU. Mohou být tedy vykresleny přímo grafickou kartou a nemusí se o to starat CPU. Jde o značné zrychlení vykreslování a odlehčení CPU. První vertex buffery - VBO (vertex

buffer object) byly dostupné již ve verzi OpenGL 1.5 v roce 2003. V dnešní době existují tyto vertex buffery:

- **VBO** - vertex buffer object - ukládání pole vertexů
- **IBO** - index buffer object - slouží pro ukládání pořadí vertexů, ve kterém mají být vykresleny
- **TBO** - texture buffer object - především určen pro textury; výhoda pro ukládání velkého množství dat
- **UBO** - uniform buffer object - pro proměnné předáváné shaderu; výhodou je jejich rychlost a hodí se pro ukládání často aktualizovaný dat

```
//Vytvoření vertex array objektu
glGenVertexArrays( 1, &TerrainVAO );
//Nastavení TerrainVAO jako aktuálního objektu
glBindVertexArray( TerrainVAO );

//Vytvoření vertex buffer objektu
glGenBuffers( 1, &TerrainVBO );
//Nastavení TerrainVBO jako aktuálního objektu
glBindBuffer( GL_ARRAY_BUFFER, TerrainVBO );
//Nastavení VBO - velikost pole vertexů, ukazatel na pole vertexů, nastevaní bufferu na statický
    - data se nebudou měnit
glBufferData( GL_ARRAY_BUFFER, sizeof(GLfloat)*vertex_count, &vertices[0], GL_STATIC_DRAW);

//Nastavení ukazatele na pozici vertexu pro shader
glEnableVertexAttribArray(0);
//ID atributu, velikost atributu, datový typ, offset atributu v poli vertexů, offset ve struktuře
glVertexAttribPointer (0, 3, GL_FLOAT, GL_FALSE, sizeof(GLfloat)*6, (void*)0);

//Nastavení ukazatele na normálu vertexu pro shader
glEnableVertexAttribArray(1);
//ID atributu, velikost atributu, datový typ, offset atributu v poli vertexů, offset ve struktuře
glVertexAttribPointer (1, 3, GL_FLOAT, GL_FALSE, sizeof(GLfloat)*6, (void*)( sizeof(GLfloat)*3));

//Vytvoření index buffer objektu
glGenBuffers( 1, &TerrainIBO );
//Nastavení TerrainIBO jako aktuálního objektu
glBindBuffer( GL_ELEMENT_ARRAY_BUFFER, TerrainIBO );
//Nastavení IBO - velikost pole indexů, ukazatel na pole indexů, nastevaní bufferu na statický -
    data se nebudou měnit
glBufferData( GL_ELEMENT_ARRAY_BUFFER, sizeof(GLuint)*indicesCount_, indices_,
    GL_STATIC_DRAW );

//Odpojení VAO
glBindVertexArray(0);
//Smazání pole vertexů - už nepotřebujeme, jsou uloženy ve VBO
delete vertices;
```

Výpis 5: Vytváření objektu VAO s VBO a IBO

Před vytvořením VAO se naplní struktury vertexů daty a pole indicií indexy jednotlivých vertexů. V aplikaci je pro terén vytvořen jeden VAO, který obsahuje VBO a IBO. VBO se předá velikost pole struktur vertexu a ukazatel na něj. IBO se předa to samé, akorát indicie jsou pole čísel typu `GLuint`. U VBO se nastaví ukazatele na pozici a normálu ve struktuře každého vertexu.

Při vykreslování se jen připojí VAO a zavolá se vykreslení trojúhelníků:

```
//Nastavení TerrainVAO jako aktuálního objektu
glBindVertexArray( TerrainVAO );

//Vykreslení trojúhelníků s počtem rovným počtu indicií (celý model), datový typ indicií, ukazatel
    na pole indicií
glDrawElements( GL_TRIANGLES, indicesCount_, GL_UNSIGNED_INT, NULL );

//Odpojení VAO
glBindVertexArray(0);
```

Výpis 6: Vytváření objektu VAO s VBO a IBO

4.3.2 Shadery

Shader je program vykonávaný grafickou kartou. Má svůj specifický programovací jazyk. Pro OpenGL je to GLSL, DirectX má HLSL. Jejich konstrukce a datové typy se mírně liší. Základní typy shaderů jsou:

- **Vertex shader**
- **Tesselation shader** - od OpenGL 4.0
- **Geometry shader**
- **Fragment shader**

Vertex shader Spuštěn pro každý vertex přicházející do GPU. Jeho funkcí je transformovat 3D pozici bodu na 2D pozici ve výsledném obrazu na obrazovce. Pracují s pozicí, barvou a souřadnicí textury každého bodu. Nemůžou vytvářet nové body. Výstup vertex shaderu jde do dalšího stupně - do geometry shaderu (případně do teselátoru).

Geometry shader Nový typ shaderu (od OpenGL 3.2). Jako vstup jsou brány celé primitiva jako body, linie a trojúhelníky. Zde mohou nová primitiva také vznikat.

Fragment shader Fragment shader nebo také pixel shader. Ovlivňuje vlastnosti každého vykresleného pixelu. Zde se provádí výpočty osvětlení a aplikace osvětlovacích modelů, výpočty stínů a změna tvaru povrchu bez modifikování vertexů (tzv. bump-mapping). Fragment shadery se používají také pro různé efekty.

5 Efektivní vykreslování scény

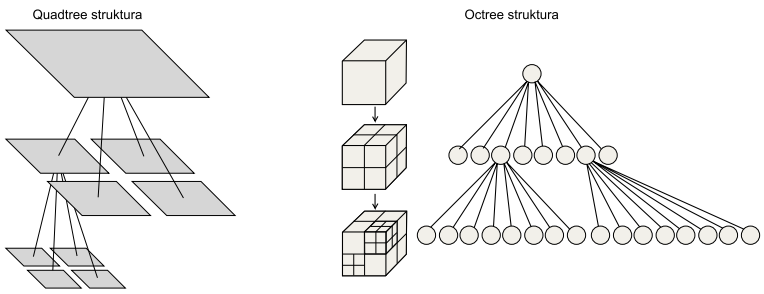
5.1 Hierarchie scény

Terén ve vytvořené aplikaci má velikost 1025x1025 vertexů, kde každá struktura vertexu obsahuje jeho pozici, normálu a souřadnice textury typu `GLfloat`. Každá struktura má tedy 32 bajtů a celý terén 32MB dat. Tyto hodnoty jsou vzhledem k dnešnímu hardwaru zanedbatelné, avšak pro vykreslování v reálném čase je třeba využít veškerých metod ke zvýšení fps a ulehčení práce jak grafické kartě, tak i procesoru.

Je tedy třeba uvažovat určitou hierarchii scény a s tím spojené řešení viditelnosti. Většina algoritmů týkajících se hierarchie scény je založena na binárním rozdělování prostoru (BSP). BSP je proces rekurzivního rozdělování scény podle daných požadavků. Seznam nejznámějších BSP stromů:

- octree
- quadtree
- k-d tree
- vp-tree

Pro scénu v exteriéru, lze využít jak octree tak i quadtree. Octree lze využít na práci s objekty ve scéně. Pro samotný terén jsem zvolil quadtree, který je o jednu dimenzi menší než octree a jedná se o rekurzivním dělení scény ve dvou osách (X a Z) vždy na čtyři části - uzly. Vychází se z kořene, což je celý terén a dělení scény končí, když stupeň uzlu quadtree dosáhne úrovně 6, což je síť o rozměrech 64 x 64 uzlů.



Obrázek 15: Ukázka quadtree a octree rozdělení scény

Každý uzel obsahuje ukazatele na své čtyři potomky, informace o své pozici ve scéně, jeho rozměrech, úrovni uzlu v quadtree, vzdálenosti od kamery a indicie vertexů v IBO. Díky souřadnicím uzlu a jeho velikosti můžeme každou část meshe reprezentovat jednoduchým primitivem - v mém případě krychlí (AABB). Reprezentace částí terénu pomocí primitiv slouží k jednoduchému určení, zda je daná skupina polygonů, připadajících uzlu, viditelná nebo ne. Ukázka programu vytvářejícího strukturu quadtree.

```
QuadTree::QuadTree( unsigned int maxLevel ) {
  this->maxLevel_ = maxLevel;
  root_ = new QuadTreeNode( 512, 512, 1024 );
  root_->level_ = 0;
  root_->parent_ = 0;
  Subdivide( root_ );
}

void QuadTree::Subdivide( QuadTreeNode* node ) {
  if ( node->level_ < maxLevel_ ) {
    unsigned short nWidth = node->width_/4;
    node->nodes_[0] = new QuadTreeNode( node->centerX_-nWidth, node->centerZ_-nWidth,
        nWidth*2 );
    node->nodes_[1] = new QuadTreeNode( node->centerX_+nWidth, node->centerZ_-nWidth,
        nWidth*2 );
    node->nodes_[2] = new QuadTreeNode( node->centerX_-nWidth, node->centerZ_+nWidth,
        nWidth*2 );
    node->nodes_[3] = new QuadTreeNode( node->centerX_+nWidth, node->centerZ_+nWidth,
        nWidth*2 );

    for( int i = 0; i < 4; i++ ) {
      node->nodes_[i]->parent_ = node;
      node->nodes_[i]->level_ = node->level_ + 1;
      Subdivide( node->nodes_[i] );
    }
  }
  else {
    for( int i = 0; i < 4; i++ ) {
      node->nodes_[i] = 0;
    }
  }
}
```

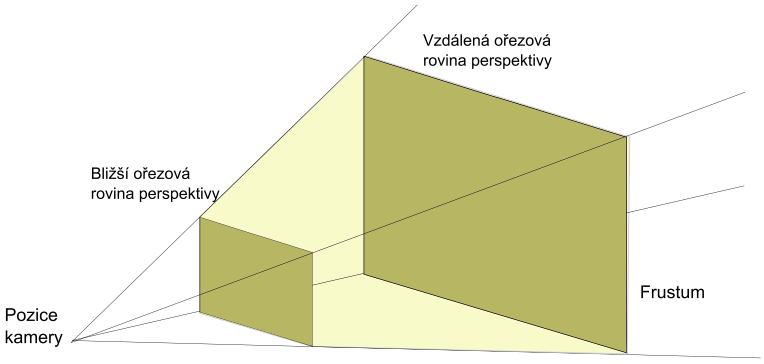
Výpis 7: Ukázka kódu při dělení uzlu

5.2 Řešení viditelnosti

5.2.1 Frustum Culling

První fází řešením viditelnosti je proces, zvaný Frustum Culling, který je zvláště efektivní u hierarchické struktury prostředí. Je to proces pro zahození uzlů, které jsou mimo zorné pole kamery. Pokud nesledujeme celý terén, je tato část týkající se viditelnosti efektivní, protože díky frustum cullingu neposíláme grafické kartě data, která stejně nakonec nebudou ve scéně viditelná. Frustum je ta část scény, kterou definuje nastavení kamery. Má tvar pyramidy s useknutým vrcholem. Je tedy definován šesti ořezovými rovinami tzv. clipping planes.

Dvě roviny, normálami směřujícími do kamery, jsou definované parametry OpenGL perspektivy `near` a `far`. Ostatní roviny jsou definované úhlem pohledu (`fovy`) a poměrem šířka/výška (`aspect`). K testování vůči frustumu potřebujeme znát `modelView`



Obrázek 16: Frustum

a `projection` OpenGL matice. Jejich vynásobením získáme `modelViewProjection` matici. Touto maticí vynásobíme vertex a jeho pozici v prostoru zvaném clip-space. Clip-space má souřadnice na osách X, Y a Z v rozmezí (-1,1). Nyní stačí touto maticí vynásobit všechny vertexy AABB testovaného uzlu a porovnat jejich souřadnice X, Y a Z, zda jsou v rozmezí od -1 do 1.

Existují 3 případy výsledku testování:

- všechny body jsou uvnitř - vykreslit uzel
- všechny body jsou mimo - zahodit uzel
- část bodů je uvnitř - otestovat potomky uzlu

Postupně procházíme všechny AABB terénu od samotného kořene quadtree k nejnižším potomkům. Když jsou všechny body AABB mimo rozmezí, je část terénu neviditelná z pohledu kamery a daný uzel můžeme zahodit a už s ním nemusíme v tomto framu počítat. Když jsou všechny uvnitř, vykreslíme daný uzel. Jinak procházíme potomky daného uzlu rekurzivně. Následující pseudokód popisuje tuto problematiku:

```
ProcessNode( node ) {
    if ( AABBInFrustum( node ) == INSIDE || node->level == maxLevel ) {
        node->visible = 1;
    }

    else if ( AABBInFrustum( node ) == PROCESS ) {
        node->visible = 2;

        foreach childNode in node->nodes {
            ProcessNode( childNode );
        }
    }

    else node->visible = 0;
}
```

Výpis 8: Pseudokód procházení quadtree struktury při Frustum Cullingu

5.2.2 Occlusion Culling

Poté, co jsme se zbavili největší části terénu frustum cullingem, přichází další, poněkud rozsáhlejší a složitější fáze. Tato fáze se nazýva Occlusion Culling a jedná se o algoritmus, který řeší viditelnost z pohledu, kdy jeden objekt zakrývá objekt druhý. V tomto případě druhý objekt nebude vykreslen, avšak bez occlusion cullingu bychom s tímto objektem počítali. Normálně je tento problém v OpenGL řešen hardwarovou konstrukcí zvanou Z-buffer. To pro naši potřebu není nejlepší řešení, protože tento mechanismus počítá všechny objekty ve scéně tak, že porovnává Z-hodnotu každého pixelu objektu s obsahem Z-bufferu a když je hodnota menší, pixel v Z-bufferu aktualizuje na danou hodnotu. Při scéně s mnoha objekty je tohle řešení pomalé. Důležitým pojmem v této kapitole je `occluder`, což je objekt, který zakrývá další objekty. Ten se volí většinou podle toho, jak velkou část vykreslované plochy zabírá. Existuje několik osvědčených řešení okluze:

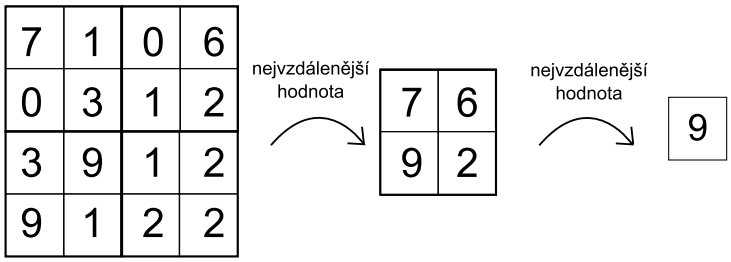
Shadow culling Algoritmus, při kterém se vybere podle určitých kritérií set occluderů. Tento set bývá většinou omezen maximálním počtem occluderů pro zachování dobrého poměru efektivity a rychlosti. Pro každý tento objekt se sestrojí occlusion volume, což je stejná věc jako shadow volume. Je to tedy část scény, kterou objekt zakrývá. Ostatní objekty se poté porovnávají v rámci této oblasti stejně jako při řešení frustum cullingu. Změna je jen v tom, že pokud je objekt uvnitř oblasti je neviditelný na rozdíl od frustumu.

Hierarchical occlusion map algoritmus HOM algoritmus [**?**] je rozdělen do dvou částí. První je 1D test v ose Z. Druhý je 2D test v osách X a Y ke zjištění, zda je objekt zakrýván occluderem. Před vykreslováním scény je vybrán vhodný set occluderů.

V první fázi vykreslování se vykreslí tyto occludery bez testování viditelnosti. Vykreslují se do color bufferu bílou barvou na černé pozadí. Při tomto vykreslování je vypnuto osvětlení, texturování a hloubkové testování (z-buffering). Tímto se ze všech occluderů vytvoří jeden obraz, zvaný okluzní mapa, podle kterého se budou testovat ostatní objekty, zda mají být vykresleny. Z této mapy se rekurzivně vytvoří pyramida okluzních map, jako v případě hierarchických z-map, ale výsledný pixel reprezentuje průměrnou hodnotu čtyř pixelů předchozí úrovně mapy. Rekurze končí, když rozlišení mapy dosáhne předem daného minimálního rozlišení. Tím, že se bere průměrná hodnota pixelů, přibývá při snižujícím se rozlišení pixelů šedých odstínů. Tyto pixely reprezentují průhlednost v daném místě.

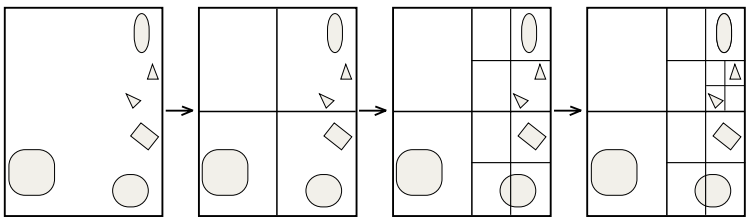
Druhá fáze, tedy test vůči okluzním mapám, promítne ohraničující tvar testovaného objektu do screen-space. Zde získá 2D reprezentaci jeho ohraničujícího tvaru - obdélník. Tento tvar pokrývá větší část než samotný objekt, proto je test konzervativní. To znamená, že pokud je objekt označen jako nezakrytý, nemusí to být pravda a objekt, aniž by byl vidět, bude vykreslen. HOM test vybere úroveň okluzní mapy, kde jeden „pixel“ mapy je přibližně velký jako obdélník reprezentující objekt. Pokud jsou všechny pixely mapy v oblasti objektu bílé, je objekt zakryt. V druhém případě je obdélník testován mapou s větším rozlišením.

Hierarchical Z-buffer algoritmus Algoritmus hierarchické viditelnosti [3]. Tento algoritmus rozděluje scénu pomocí octree a využívá Z-bufferu, ze kterého vytváří několik stupňů od původního nejvyššího rozlišení (původní Z-buffer), až po rozlišení 1x1, kde každý stupeň obrazu má vždy čtvrtinové rozlišení předchozího stupně (2x2 pixely předchozího stupně tvoří pixel stupně dalšího). Při generování každého stupně se výsledná hodnota Z-bufferu rovná maximální hodnotě ze čtyř pixelů předchozího rozlišení. Je to tedy vždy nejvzdálenější hodnota v Z-bufferu. Těchto stupňů Z-bufferu se využívá pro testování jednotlivých objektů ve scéně, zda jsou viditelné nebo ne.



Obrázek 17: Proce vytváření hierarchických Z-map (převzato z [8])

Nejdříve se sestrojí octree struktura, do které se postupně uloží všechny objekty scény. Konstrukce stromu je časově náročnější, proto se provádí pouze při startu programu a hodí se tedy jen pro statické objekty. Při organizování jednotlivých objektů do octree struktury využívá Greene algoritmus, který se vyhýbá přiřazování malých objektů do velkých uzlů octree. Opět se začíná kořenem octree, který obklopuje celou scénu jedním AABB, který se rekurzivně dělí na 2x2x2 AABB. V každém kroku rekurze se ověřuje, jestli daný uzel (box) neobsahuje množství objektů menší, než předem daná hodnota. Pokud je tato podmínka splněna, objekty se přiřadí danému boxu do jeho pole ukazatelů. Když je množství objektů větší, pokračuje se v rekurzi dělení boxu na menších osm. Celý proces je ukončen, když je podmínka množství objektů pro každý uzel octree splněna, nebo dokud se nedosáhlo určitého stupně rekurze.



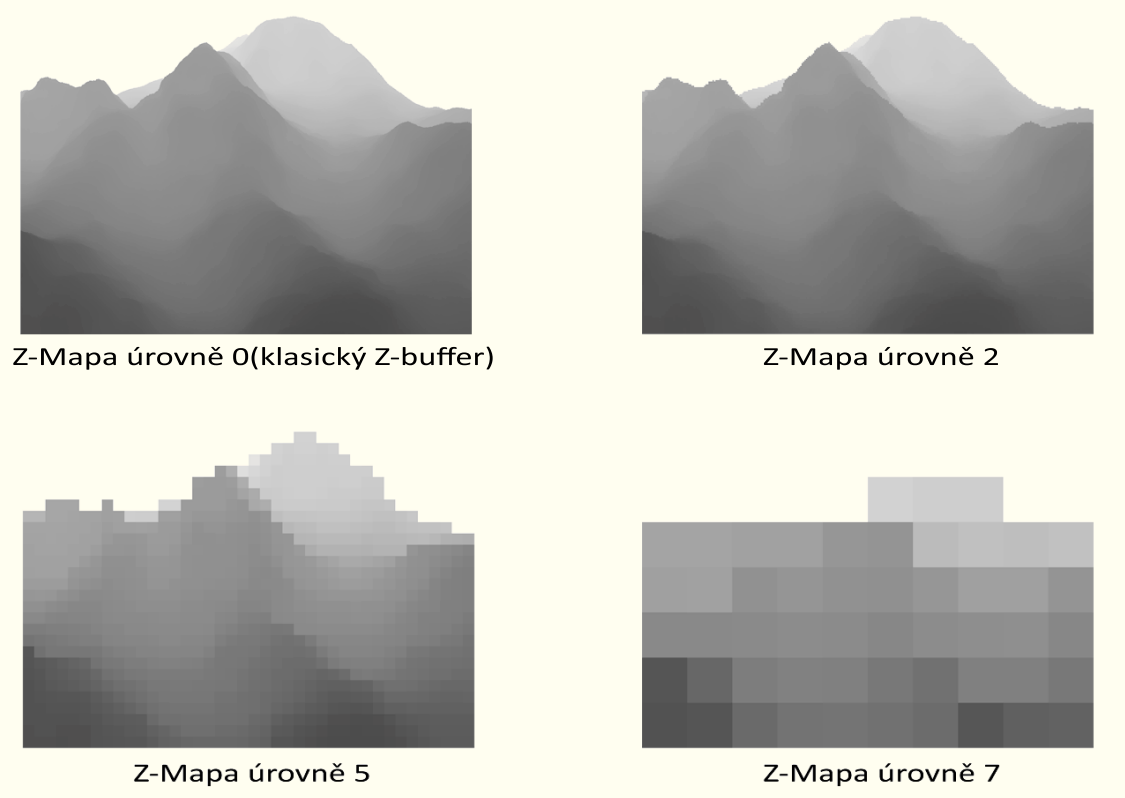
Obrázek 18: Quadtree pro objekty podle Greena (převzato z [8])

Po přiřazení všech objektů je octree kompletní a je připraven pro vykreslování. Nejdříve se na octree použije frustum culling, pro zbavení se uzlů mimo naši sledovanou oblast. Poté se rekurzivně prochází octree od kořene a zjišťuje se, jestli je AABB konkrétního uzlu viditelný vzhledem k Z-pyramidě. Jestli AABB neprojde testem, je vyřazen a objekty

jemu přiřazené nebudou vykresleny. Když teste projde, objekty jsou vykresleny, aktualizuje se Z-buffer a rekurzivně se otestují potomci uzlu. Po skončení rekurze je obraz Z-bufferu hotov.

Proces zjištění, zda je uzel octree viditelný je následující. Porovnáme všechny strany AABB se Z-pyramidou. V pyramidě najdeme úroveň takovou, že její „buňka“ obklopuje celou stranu AABB. Získáme nejbližší Z souřadnici dané strany a porovnáme s hodnotou v Z-pyramidě (hodnota dané buňky). Pokud je hodnota v buňce Z-bufferu menší, než Z souřadnice strany, je strana neviditelná. Takto se projdou všechny strany AABB a zjistí se viditelnost celého uzlu. Výhoda Z-pyramidy je v tom, že vzhledem ke klasickému Z-bufferu je počet nutných testů ke zjištění viditelnosti mnohonásobně menší.

Toto řešení je použito i v aplikaci. Řešení založeno na implementaci popsané zde [9].



Obrázek 19: Příklady Z-map v aplikaci

Occlusion queries Occlusion query je hardwarové řešení viditelnotsi. Byla to jedna z nejvíce očekávaných hardwarových novinek. Díky této vlastnosti můžeme ještě před samotným vykreslením objektu zjistit, zda bude na výsledné obrazovce vykreslen. Pošleme GPU zjednodušený model testovaného objektu, nejčastěji bounding box, a GPU nám vrátí počet pixelů, které by byly vykreslené na obrazovce. Podle této hodnoty se poté

můžeme rozhodnout, zda objekt vykreslit nebo ne. To nám může značně usnadnit práci, ale i toto řešení není samo o sobě dokonalé.

Postup při implementaci occlusion query:

1. Vytvoříme query
2. Zakážeme vykreslování a do Z-bufferu
3. Spustíme query
4. „Vykreslíme“ bounding box objektu (provede se test hloubky).
5. Ukončíme query
6. Povolíme vykreslování a do Z-bufferu
7. Získáme výsledek query (počet viditelných pixelů).
8. Pokud je počet pixelů větší než určitá hodnota, nejčastěji 0, vykreslíme objekt

Tvar objektu v kroku 4 by měl být co nejjednodušší, aby byl test co nejkratší, ale měl by pokrýt minimálně stejnou oblast na obrazovce jako objekt původní. Tato metoda je zvláště efektivní u složitých objektů. Největší problém occlusion queries je právě v kroku 7, kdy musíme čekat na výsledek.

Tento problém je způsoben pipeline v GPU. Při běžných akcích pracují CPU a GPU paralelně. CPU pošle grafickému procesoru příkaz, aby něco vykreslil a dále se o to nestará. Takhle neustále posílá příkazy a pokračuje ve své práci. Příkazy se v GPU řadí do fronty a až na ně dojde řada, tak jsou vykonány. Při occlusion query je průběh ovšem jiný.

Protože CPU očekává výsledek query, musí nejdříve GPU vykreslit předchozí příkazy v pipeline, než se dostane k příkazu, kterého se query týká. Poté se musí daný objekt vykreslit a až nakonec GPU vrací počet pixelů, které prošly testem. Tohle čekání CPU na GPU, při ulehčování práce GPU, nakonec může celý vykreslovací proces dokonce zpomalit.

Jedno vylepšení spočívá ve spuštění více queries najednou a vyzvednutí výsledků po skončení všech testů:

1. Vytvoříme n queries
2. Zakážeme vykreslování a do Z-bufferu
3. Foreach query
   - Spustíme query
   - „Vykreslíme“ bounding box objektu (provede se test hloubky).
   - Ukončíme query
4. End foreach query

5. Povolíme vykreslování a do Z-bufferu
6. Foreach query
    - Získáme výsledek query (počet viditelných pixelů).
    - Pokud je počet pixelů větší než určitá hodnota, nejčastěji 0, vykreslíme objekt
7. End foreach query

V tomto případě je efektivita o něco lepší, ale pořád se musí čekat na výsledek, což zbytečně zpomaluje vykreslování. V praxi se většinou mezi ukončením query a získáním výsledku testování využívá CPU k vykonání jiné práce. Procesor tak místo čekání může dělat práci, která by stejně měla být vykonána a až ji dodělá, tak požádá GPU o výsledky queries. Tohle řešení už je dokonalejší a přijatelné.

5.2.3 Problémy spojené s bounding objekty

Pro testování vůči frustumu je nejlepší volbou bounding sphere, protože se testuje poloha jejího středu s offsetem o velikosti poloměru koule. Oproti tomu je porovnávání osmi vrcholů krychle pomalé. V některých situacích je ale koule jako bounding volume nepoužitelná vzhledem k jejímu tvaru a tvaru objektu. Problém vznikne u dlouhých úzkých objektu (tyče, kabely, sloupy). V těchto případech by koule zabírala velkou plochu a navíc by tvar neodpovídal původnímu objektu a testováni by bylo nepřesné. V těchto případech je lepší použít bounding box.

Další problém je u objektů, které jsou tvarově složité. Pro správnou funkci frustum culingu je lepší tyto objekty rozdělit na více částí a každé přiřadit vlastní bounding volume.

U animovaných objektů je určování bounding objektů složitější:

- **Počítání bounding objektu každý krok animace** - v tomto případě může být počítání bounding objektu složitější než samotné vykreslení objektu
- **Vytvoření jednoho bounding objektu pro všechny kroky animace** - nejjednodušší a nejrychlejší způsob. Provede se animace objektu bez translace, aby byl bounding objekt co nejmenší, a pro každý snímek se zjistí bounding objekt. Z těchto se vytvoří výsledný. Problém tohoto řešení je v tom, že i bez translace objektu může být výsledný bounding objekt několikanásobně větší než původní objekt.
- **Uložení bounding objektu pro animace** - nejlepší řešení. Při více animacích najednou je třeba vypočítat spojení bounding objektů pro každou animaci.

Nejlepší je zkombinovat druhý a třetí přístup.

6 Úroveň detailů

Úroveň detailů je důležitou optimalizací vykreslovacího procesu. Je například zbytečné vzdálené objekty, zabírající ve výsledku pár pixelů obrazu, vykreslovat detailně a počítat s každým bodem modelu, když na výsledný obraz a dojem nebudou mít téměř žádný vliv. Proto je třeba s rostoucí vzdáleností snižovat komplexitu objektu. Také textury není nutné mít stejné kvality pro různě vzdálené objekty. Metoda změny rozlišení textur zvaná mip-mapping je známa dlouho. Funguje na podobném principu jako generování hierarchie Z-map (5). Jediný rozdíl je v tom, že při vytváření nižšího rozlišení se neuvažuje nejvzdálenější pixel ze čtveřice, ale jejich průměrná hodnota.

Jedná se o rozměrné téma a algoritmů na řešení tohoto problému existuje mnoho. Záleží i na aplikaci, které konkrétní řešení se hodí více a které méně. Dva základní přístupy jsou:

- **Diskrétní LOD (DLOD)** - základem tohoto přístupu je vytvoření více modelů s různými detaily pro jeden objekt; s rostoucí vzdáleností se vybírá model s nižší složitostí
- **Navazující LOD (CLOD)** - navazující úroveň detailů používá objekt stejný model pro jakoukoli pozici ve scéně; model objektu je zjednodušován v závislosti na vzdálenosti pomocí shaderů a tím dochází k plynulému přechodu mezi úrovněmi detailů

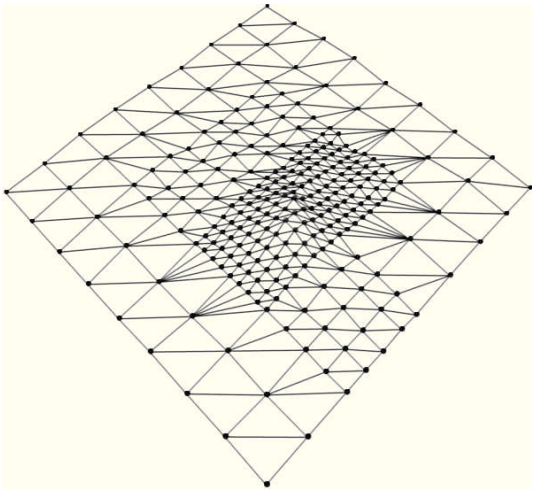
6.1 Algoritmy řešící úroveň detailů

6.1.1 ROAM

ROAM (z angl. Real-time optimally adapting meshes) je založen na dynamickém počítání trojúhelníkové sítě při každé změně pohledu ve scéně. Zachovává pevně daný počet trojúhelníků. Pracuje na principu rozdělování a sjednocení trojúhelníků tvořících síť objektu. Těmito operacemi se z jednoho trojúhelníku stanou dva, které jsou jeho potomky, nebo naopak ze dvou se stane jeden trojúhelník. Tímto rozdělováním sítě modelu se vytváří struktura binárního stromu.

6.1.2 Geometrical mipmapping

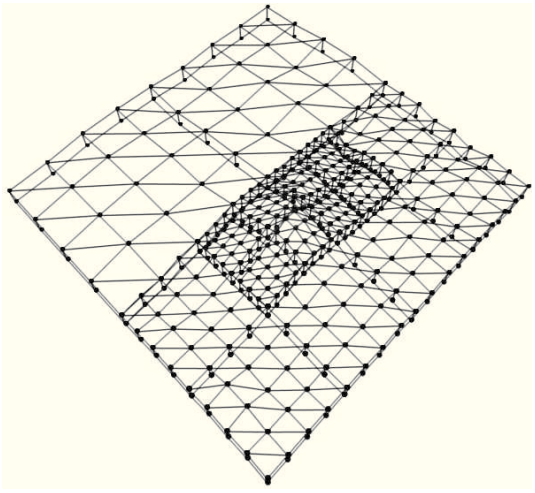
Založen na mipmapovaní terénu. Quadtree struktura ukládání terénu. Body terénu jsou uloženy ve VBO. Každá úroveň detailů (úroveň v quadtree) má uloženou topologii, které body terénu budou zobrazeny pomocí vlastního pole indexů (IBO). Každý blok terénu je poté, podle vzdálenosti od kamery, nahrazován blokem s příslušnou úrovni detailů. Problém nastane, když se potkají dva bloky, které mají různě detailní sítě. Jeden blok terénu bude mít o bod navíc na každé hraně trojúhelníku. Tím vznikají mezery v terénu. Příklad řešení rozdílu mezi dvěmi úrovněmi detailů:



Obrázek 20: Přidání hran mezi rozdílnými úrovněmi detailů terénu (převzato z [16])

6.1.3 Chunked LOD

Toto řešení je podobné předchozímu. Rozdíl je v tom, že každá úroveň detailů má uloženou vlastní síť bodů. Opět vznikají mezery v terénu, které se většinou řeší tzv. skirts. Je to řešení, kdy se k lemu každého bloku přidají další body. Ty mají stejnou pozici jako lemující body, jen souřadnice v ose Y (výška) mají hluboko pod úrovní terénu.



Obrázek 21: Skirts mezi rozdílnými úrovněmi detailů terénu (převzato z [16])

Já jsem použil pro řešení úrovní detailů hardwarovou teselaci (kapitola 4.2). Toto řešení je poměrně nové a jednoduše aplikovatelné. Také jsem chtěl teselaci vyzkoušet v praxi.

7 Závěr

Cílem práce bylo vytvořit část herního systému, konkrétně optimalizaci vykreslování. Aktuální herní systém obsahuje vygenerovaný terén pomocí šumu, který je dále vyhlazen. Terén je otexturován pomocí dvou textur pro různé strmosti terénu. Je opticky vyhlazen a vystínován pomocí vypočtených normál. Pro optimalizaci vykreslování je model terénu je rozdělen do hierarchické struktury a je aplikováno řešení viditelnosti, které odstraňuje části terénu ležící mimo zorné pole kamery. Řeší viditelnost objektů, které jsou zakryté terénem. Terén má základní síť, která je zjemňována hardwarovou teselací. Tím jsou vytvořeny úrovně detailů závislé na vzdálenosti od pozorovatele. Dále je v programu možnost zobrazení aktuálního hloubkového bufferu.

Během vypracování této práce jsem se naučil základy vytváření terénu, přístupy řešení viditelnosti a efektivity vykreslování. Prošel jsem si novými postupy při vytváření základního okna a nastavení scény. Použil jsem i jednu z novějších OpenGL technologií, kterou je hardwarová teselace. Některé části nejsou kompletní tak, jak bych si představoval nebo nejsou řešeny nejlepším způsobem a pro pokročilé v počítačové grafice může práce vypadat jednoduše. Pro začátečníka je to ale velký krok, během kterého jsem se naučil mnoho věcí. Proto bych se této práci chtěl věnovat dále a postupně ji zdokonalovat a rozvíjet. Rád bych z tohoto subsystému časem vytvořil funkční hru a poznal tak všechny části při vytváření kompletního herního systému.

8 Reference

[1] Gregory, J. *Game Engine Architecture*. 2009. ISBN 9781568814131.

[2] Wright, R. S., Lipchak, B., Haemel, N. S. *OpenGL(R) SuperBible: Comprehensive Tutorial and Reference*. Fifth edition. 2010. ISBN 0321712617.

[3] Greene, N., Kass, M., Miller, G. *Hierarchical Z-Buffer Visibility*. 1993.

[4] Shopf, J., Barczak, J., Oat, Ch., Tatarchuk, N. *March of the Froblins: Simulation and Rendering Massive Crowds of Intelligent and Detailed Creatures on GPU*. 2008.

[5] Olsen, J. *Realtime Procedural Terrain Generation*. 2004.

[6] Riaboy, J. *Real Time Rendering of Complex Height Maps*. 2003.

[7] Rollin, P., Oster, B. *OpenGL Tessellation For Professionnal Applications*. San Jose, 2010.

[8] Haines, E. Occlusion Culling Algorithms. 1999. Dostupné z URL: <http://www.gamasutra.com/view/feature/3394/occlusion_culling_algorithms.php>.

[9] Rákos, D. Hierarchical-Z map based occlusion culling. 2010. Dostupné z URL: <http://rastergrid.com/blog/2010/10/hierarchical-z-map-based-occlusion-culling/>.

[10] Sekulic, D. 2007. Efficient Occlusion Culling. Dostupné z URL: <http://http.developer.nvidia.com/GPUGems/gpugems_ch29.html>.

[11] Terrain LOD: Runtime Regular-Grid Algorithms. Dostupné z URL: <http://vterrain.org/LOD/Papers/>.

[12] Game engines. Dostupné z URL: <http://en.wikipedia.org/wiki/Game_engine>. Poslední úpravy 23.4.2012.

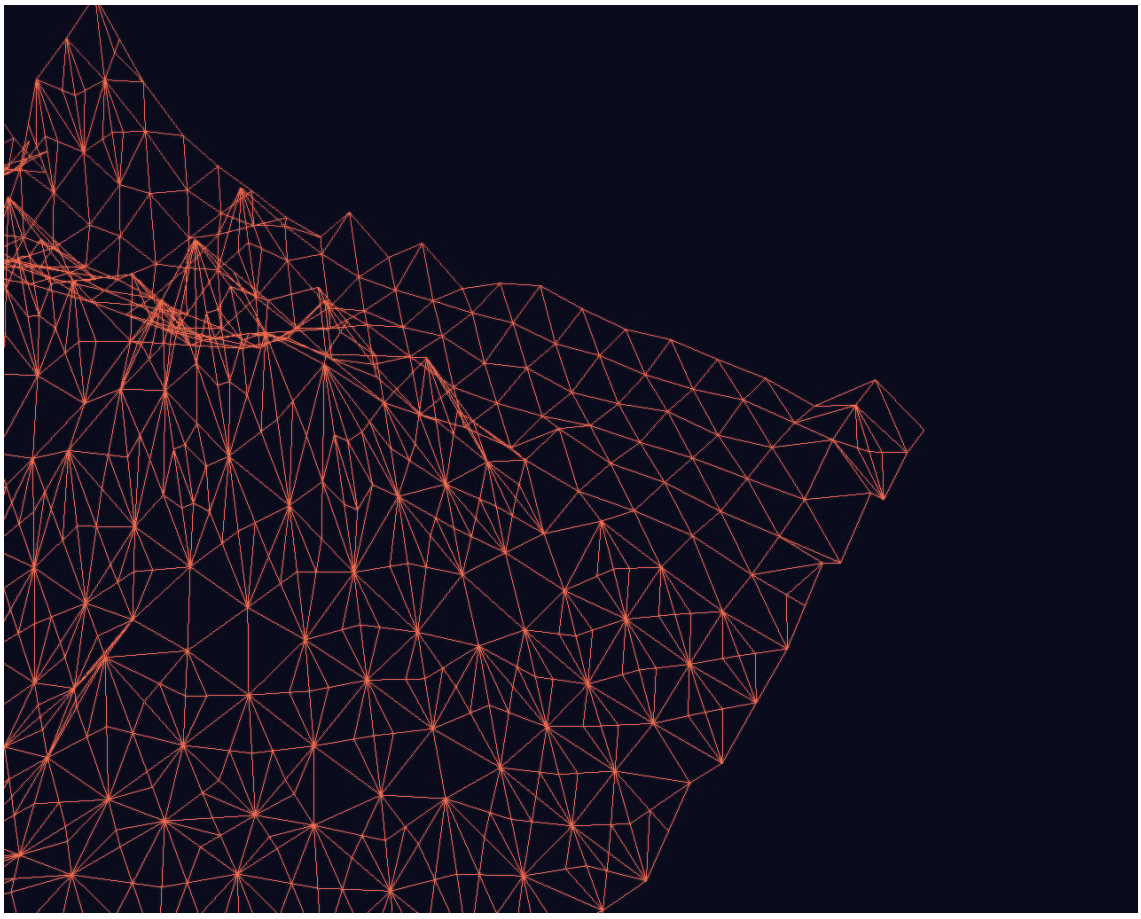
[13] Doom engine. Dostupné z URL: <http://en.wikipedia.org/wiki/Doom_engine>. Poslední úpravy 30.4.2012.

[14] Perlin noise. Dostupné z URL: <http://freespace.virgin.net/hugo.elias/models/m_perlin.htm>.

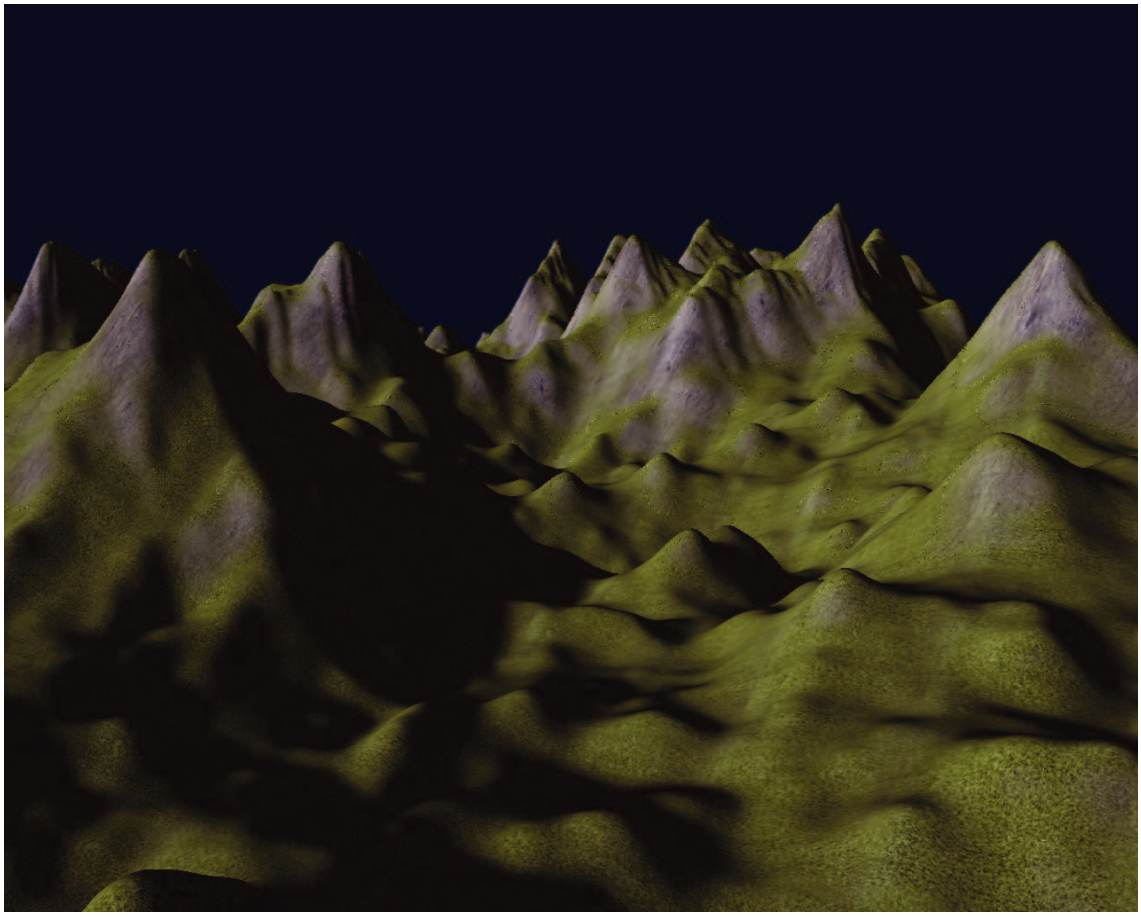
[15] Lighthouse Terrain Tutorial. Dostupné z URL: <http://www.lighthouse3d.com/opengl/terrain/>.

[16] Bartoň, R. Modern Algorithms for Real-Time Terrain Visualization on Commodity Hardware. Dostupné z URL: <http://geoinformatics.fsv.cvut.cz/gwiki/Modern_Algorithms_for_Real-Time_Terrain_Visualization_on_Commodity_Hardware>. Poslední úpravy 15.5.2011.

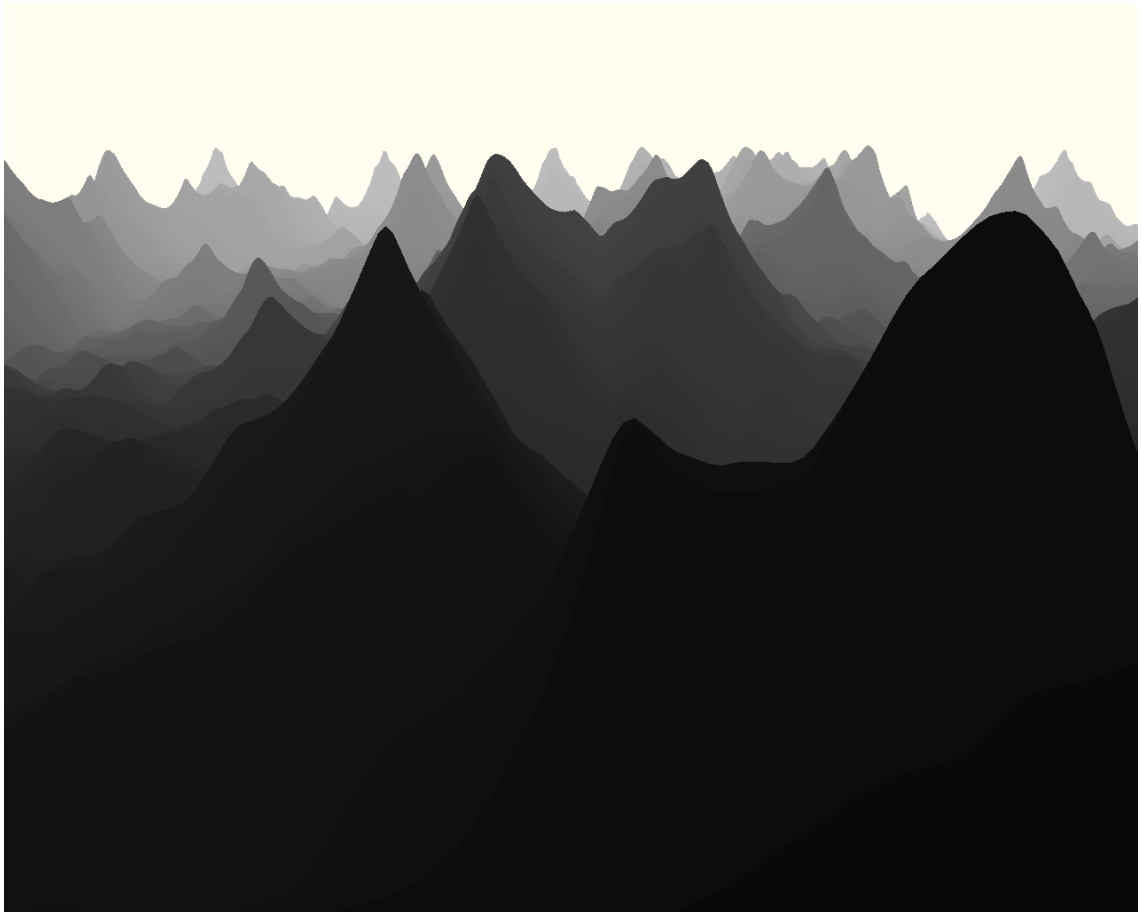
9 Ukázky výsledné aplikace



Obrázek 22: Teselace terénu



Obrázek 23: Terén v aplikaci



Obrázek 24: Obsah Z-bufferu